Towards Dynamic Term Size Computation via Program Transformation

P. López García M. Hermenegildo
pedro@dia.fi.upm.es herme@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - Spain

Abstract

Knowing the size of the terms to which program variables are bound at run-time in logic programs is required in a class of applications related to program optimization such as, for example, recursion elimination and granularity analysis. Such size is difficult to even approximate at compile time and is thus generally computed at run-time by using (possibly predefined) predicates which traverse the terms involved. We propose a technique based on program transformation which has the potential of performing this computation much more efficiently. The technique is based on finding program procedures which are called before those in which knowledge regarding term sizes is needed and which traverse the terms whose size is to be determined, and transforming such procedures so that they compute term sizes "on the fly". We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria. We also discuss the advantages and present some applications of our technique.

Keywords: Term Size Computation, Granularity Analysis, Parallelism.

1 Introduction

The need to know the size of the terms to which program variables are bound at run-time in logic programs arises in a class of applications related to program optimization which includes *recursion elimination* [16, 1], *granularity analysis* [8], and selection among different algorithms or control rules whose performance may be dependent on such size. By term size we refer to measures such as list length, term depth, number of nodes in a term, etc.

For example, in *granularity analysis* the objective is to determine (or bound) a priori (i.e. before its execution) the number of steps that the execution of a given goal will involve. Granularity analysis for a set of non recursive procedures is relatively straightforward. However, recursive procedures are somewhat more problematic: the amount of work done by a recursive call depends on the depth of recursion, which in turn depends on the size of the input. Reasonable estimates for the granularity of recursive predicates can thus be made only with some knowledge of the size of the input. In [8] a technique was presented for solving this problem based on performing a compile-time analysis which reduces granularity analysis work at run-time to evaluating simple functions of term sizes. However, the actual determination of those sizes in order to evaluate such functions is necessarily postponed until runtime. The same considerations apply in the case of recursion elimination: provided the sizes of certain terms are known a recursive predicate can be converted to a much more efficient non-recursive predicate which contains the bodies of the different recursions. Approaches such as reform compilation [16, 1] attempt to do this efficiently by performing certain preprocessing at compile-time but necessarily leave the final term size computation for run-time.

The postponement of accurate term size computation to run-time appears inevitable in general since even sophisticated compile-time techniques such as abstract interpretation are based on computing approximations of variable substitutions for generic executions corresponding to general classes of inputs, while size is however clearly a quite specific characteristic of an input. Although the approximation approach can be useful in some cases we would like to tackle the more general case in which actual sizes have to be computed dynamically at run-time.

Of course computing term sizes at run time is quite simple but at the same time it can involve a significant amount of overhead. This overhead includes both having to traverse significant parts of the term (often the entire term) and the counting process done during this traversal.

The objective of this paper is to propose a more efficient way of computing such sizes. The essential idea is based on the observation that terms are often already traversed by procedures which are called in the program before those in which knowledge regarding term sizes is needed, and thus that such sizes can often be computed "on the fly" by the former procedures after performing some transformations to them. While the counting overhead is not eliminated, overhead is reduced because additional traversals of terms are not needed. We present a systematic way of determining whether a given program can be transformed in order to compute a given term size at a given program point without additional term traversal. Also, if several such transformations are possible our approach allows finding minimal transformations under certain criteria.

The rest of the paper proceeds as follows: Section 2 introduces some of the terms to be used throughout the paper. Section 3 then presents an overview of the approach. Section 4 introduces our basic representations and Section 5 presents our concept of allowable transformations. Section 6 then introduces the concepts of irreducible and optimal transformations and highlights their important role. Section 7 then presents algorithms for finding irreducible transformations and presents an example of the complete process. Section 8 discusses the advantages of the approach, while Section 9 discusses some possible applications in more detail. Finally Section 10 presents our conclusions and suggestions for future work. We have omitted proofs for the sake of conciseness. They can be found in [10].

2 Preliminaries

This section presents some of the basic concepts to be used throughout the paper, such as term size and size relations between terms. The definitions generally follow [8].

Various measures can be used to determine the "size" of a term, e.g., term-size, term-depth, list-length, integer-value, etc. The measure(s) appropriate in a given situation can generally be determined by examining the operations used in the program. Let $|\cdot|_{\mathbf{m}} : \mathcal{H} \rightarrow \mathcal{N}_{\perp}$ be a function that maps ground terms to their sizes under a specific measure $\mathbf{m}$, where $\mathcal{H}$ is the Herbrand universe, i.e. the set of ground terms of the language, and $\mathcal{N}_{\perp}$ the set of natural numbers augmented with a special symbol $\perp$, denoting "undefined". Examples of such functions are "list_length", which maps ground lists to their lengths and all other ground terms to $\perp$; "term_size", which maps every ground term to the number of constants and function symbols appearing in it; "term_depth", which maps every ground term to the depth of its tree representation; and so on. Thus, $|[\mathtt{a},\mathtt{b}]|_{\mathrm{list\_length}} = 2$, but $|\mathbf{f}(\mathbf{a})|_{\mathrm{list\_length}} = \perp$. Given a set of terms $\mathbf{S}$, a substitution θ is said to be *S-grounding* if $\theta(\mathbf{t})$ is a ground term for every term $\mathbf{t}$ in $\mathbf{S}$. The function $size_{\mathbf{m}}(\mathbf{t})$ defines the size of a term $\mathbf{t}$ under the measure $\mathbf{m}$:

$$size_{\mathbf{m}}(\mathbf{t}) = \begin{cases} \mathbf{n} & \text{if } |\theta(\mathbf{t})|_{\mathbf{m}} = \mathbf{n} \text{ for every} \\ & \{\mathbf{t}\}\text{-grounding substitution } \theta \\ \perp & \text{otherwise.} \end{cases}$$

The function $diff_{\mathbf{m}}(\mathbf{t}_1, \mathbf{t}_2)$ gives the size difference between two terms $\mathbf{t}_1$ and $\mathbf{t}_2$ under the measure $\mathbf{m}$:

$$\mathit{diff}_{\mathbf{m}}(\mathbf{t}_1,\mathbf{t}_2) = \begin{cases} \mathbf{d} & \text{if } |\theta(\mathbf{t}_2)|_{\mathbf{m}} - |\theta(\mathbf{t}_1)|_{\mathbf{m}} = \mathbf{d} \\ & \text{for every } \{\mathbf{t}_1,\mathbf{t}_2\}\text{-grounding} \\ & \text{substitution } \theta \\ \\ \bot & \text{otherwise.} \end{cases}$$

Thus,

$\mathit{diff}_{\text{list\_length}}(\texttt{[c | L]},\texttt{[a, b | L]}) = 1,$
$\mathit{diff}_{\text{term\_depth}}(\texttt{f(a, g(X)), X}) = -2,$
$\mathit{diff}_{\text{term\_depth}}(\texttt{f(X, Y), X}) = \bot.$

In the discussion that follows we will omit the subscript in the *size* and *diff* functions when the particular measure under consideration is clear from the context.

As an example of the size analysis proposed in [8], consider the predicate `nrev/2`, defined as:

```
nrev([], []).
nrev([H|L], R)  :-  nrev(L, R1), append(R1, [H], R).
```

Let **head**[**i**] denote the size of the term appearing at $\mathbf{i^{th}}$ argument position in the head and $\mathbf{body_j}$[**i**] in the $\mathbf{j^{th}}$ literal of the body. Using *size* and *diff* functions, size analysis get the following argument size relations between terms appearing in arguments positions of the second clause:

$\mathbf{body}_1[1] = \mathbf{head}[1] + \mathit{diff}(\texttt{[H | L]},\texttt{L}),$
$\mathbf{body}_2[1] = \mathbf{body}_1[2] + \mathit{diff}(\texttt{R1},\texttt{R1}),$
$\mathbf{body}_2[2] = \mathit{size}(\texttt{[H]}),$
$\mathbf{head}[2] = \mathbf{body}_2[3] + \mathit{diff}(\texttt{R},\texttt{R}).$

For the first clause: $\mathbf{head}[1] = \mathit{size}([])$, and $\mathbf{head}[2] = \mathit{size}([])$.

Using *list-length* as measure, and after a normalization process, size analysis can infer the following size relations for the second clause:

$\mathbf{body}_1[1] = \mathbf{head}[1] - 1,$
$\mathbf{body}_2[1] = \mathbf{body}_1[2],$
$\mathbf{body}_2[2] = 1,$
$\mathbf{head}[2] = \mathbf{body}_2[3].$

and for the first one: $\mathbf{head}[1] = 0$, and $\mathbf{head}[2] = 0$.

3 Overview of the Approach

As mentioned in the introduction, we are interested in transforming some predicates in such a way that they will compute some of their argument data sizes at run-time, in addition to performing their normal computation. It is often the case that because of previous transformations or other reasons the size of certain terms is already known and it can be used as a starting point in the dynamic computation of those that we need to determine at a given point. Thus, we will be interested in the general problem of transforming programs to determine the sizes of one set of terms given that the sizes of the terms in another (disjoint) set are known.

Example 3.1 Consider, for example, the predicate `append/3`, defined as:

```
append([],L,L).
append([H|L],L1,[H|R])  :-  append(L,L1,R).
```

We can transform this predicate in such a way that it computes the size of the third argument, provided that the size of the second one is supplied. The transformed predicate can be defined as:

```
append3i2([],L,L,S,S).
append3i2([H|L],L1,[H|R],S2,S3)  :-  append3i2(L,L1,R,S2,Sb3), S3 is Sb3 + 1.
```

where the fourth and fifth arguments of this predicate are the sizes of the second and third respectively.[1] □

To perform this transformation, size relations between terms appearing in each clause have to be known. To transform the first clause the size relation $\mathbf{head}[3] = \mathbf{head}[2]$ is needed. For the second one the following size relations are needed: $\mathbf{head}[3] = \mathbf{body}_1[3] + 1$, and $\mathbf{body}_1[2] = \mathbf{head}[2]$.

In this case $\mathbf{body}_1[3]$ is recursively computed, and $\mathbf{body}_1[2]$, which is needed for this computation, is supplied by the head. The transformed predicate, `append3i2/5`, performs the same computation as `append/3`, while in addition computing the size of the third argument of `append/3` as a function of the size of the second argument.

To perform the transformation it is necessary to know for each clause, and for each term occurring at a head position whose size is going to be computed by the transformed procedure at run-time, an expression which gives the size of the term as a function of the sizes of other positions in the clause.

4 Transforming Procedures: *Transformation Nodes*

In this section we explain how the information needed for procedure transformation is represented. We also formulate some conditions that this information has to fulfill in order for the transformation to lead to correct size computations. We thus prepare the way to end with the definition of a *transformation node*, which can be considered as a data structure which contains the information needed to transform a procedure. Transformation nodes will also later be nodes in a search tree when the algorithm used to find different forms of transforming sets of procedures, or whole programs is presented.

definition 1 (Procedure Transformation Label (PTL)) *a structure,* $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$*, where:*

- **Pred***: is the name and arity of the predicate to be transformed.*
- **Os***: is a list of argument positions (represented as numbers) whose sizes are computed by the transformed predicate at run-time.*
- **Is***: is a list of argument positions whose size are needed to compute the size of argument positions in* **Os***. These sizes must be supplied by previous computations.*

The condition: $\mathbf{Os} \cap \mathbf{Is} = \emptyset$ *is required.* □

With the above defined labels we can express which predicate **Pred** is transformed and which argument sizes will be computed as a function of which others. Transformation nodes will be labeled with such PTLs. An example of a PTL is: $\mathbf{ptl}(\mathbf{append}/3, [3], [2])$;which states that the predicate $\mathbf{append}/3$ will be transformed to compute the size of its third argument, provided that the size of the second one is supplied at the procedure call. This means that it is necessary to add two extra arguments to the transformed predicate which will stand for the sizes of the second and third argument of $\mathbf{append}/3$.

definition 2 (Term Size Descriptor (TSD)) *a structure of the form:*

[1] For clarity, this class of transformations is used in the examples even if they are not ideal given that they destroy tail recursion optimization. However it is quite straightforward to perform the equivalent transformation which preserves tail recursion optimization by using an accumulating parameter. These are the transformations performed in practice.

$$\mathbf{tsd(ptl(Pred, Os, Is), ArNum, LitNum, [Exp1, .., ExpN])}$$

where:

- $\mathbf{ptl(Pred, Os, Is)}$ *is a procedure transformation label;*
- **LitNum***: is a literal number in a clause (literals are numbered from left to right, starting by assigning one to the literal after the head);*
- **ArNum***: is an argument number of literal* **LitNum***; and,*
- $\mathbf{Exp1, .., ExpN}$ *: are* Valid Size Expressions*, to be defined shortly.*

The condition: $\mathbf{ArNum \in Os}$ *is required.* □

A *TSD* describes the size of a term appearing in a body clause. It supplies information about the position in the body at which the term occurs, what sizes are computed by the literal, and which are the terms whose size is needed for this computation. **LitNum** and **ArNum** give the position in the body of the clause at which the term whose size is described occurs. The condition states that the size required has to be computed by the transformed literal. $\mathbf{ptl(Pred, Os, Is)}$ describes the size computation for which the literal **LitNum** is transformed. $\mathbf{Exp1, .., ExpN}$ describe the sizes of the terms that occur at arguments of the literal number **LitNum** in **Is**. These sizes are needed for the transformed literal to perform the computation of sizes.

An example of TSD may be: $\mathbf{tsd(ptl(append/3, [3], [2]), 3, 1, [h(2)])}$. This represents that the size of the third argument of $\mathbf{append/3}$ is computed by the transformed literal number 1, and states that the size of its second argument, needed for this computation, is supplied by the second argument of the head ($\mathbf{h(2)}$).

definition 3 (Size Expression) *A* size expression *is recursively defined as follows:*

- *A Natural number is a* size expression.
- *A term* $\mathbf{h(i)}$*, is a* size expression.
- *A Term Size Descriptor (TSD) is a* size expression.
- *If* $\mathbf{E_1}$ *and* $\mathbf{E_2}$ *are* size expressions*, then* $\mathbf{E_1 \triangle E_2}$ *is a* size expression*, where* $\triangle$ *is any usual arithmetic operator (+, −, ∗, exponentiation, etc.).*

$\mathbf{h(i)}$ *denotes argument number* **i***, or position number* **i** *of a clause head.* □

definition 4 (Valid Size Expression) *a size expression* **Exp** *is valid if it meets the following conditions:*

1. *For each term size descriptor:*

   $$\mathbf{tsd(ptl(Pred, Os, Is), ArNum, LitNum, [Exp1, .., ExpN])}$$

   appearing in **Exp***, and for each literal number* **n** *appearing in the term size descriptors of* $\mathbf{[Exp1, .., ExpN]}$*,* $\mathbf{n < LitNum}$*.*

2. *All size relations expressed in* **Exp** *are valid (we refer to size relations, as those described in section 2). A size relation is valid if it is true for every substitution that makes the terms occurring in such size relation ground.* □

A valid size expression provides information about the size of some term in a clause. If such an expression is a TSD then it expresses which body literal computes the size, and the size expressions that appear in the TSD provide the size of the arguments needed for this size computation. If the valid size expression is a head position ($\mathbf{h(i)}$), then it represents the size of the $\mathbf{i^{th}}$ argument of the head.

Condition 1 says that the sizes supplied to a transformed literal can be computed only by previous literals of the body. This requirement is due to the fact that the sizes supplied have to be "ground" at the call, because we are interested in using built-ins similar to "is/2" (in fact, more efficient and specialized versions) to perform the arithmetic operations needed to compute sizes and these built-ins require all but one of their arguments to be ground.

An example of a valid size expression, taken from Example 3.1 is:

$$1 + \mathbf{tsd}(\mathbf{ptl}(\mathbf{append}/3, [3], [2]), 3, 1, [\mathbf{h}(2)])$$

which states that in the expression $1 + \mathbf{body}_1[3]$, $\mathbf{body}_1[3]$ can be computed by literal number 1, provided that $\mathbf{body}_1[2]$ is supplied, and that $\mathbf{body}_1[2] = \mathbf{head}[2]$, i.e., that it is in fact supplied by the head.

Once we have all necessary definitions we define the concept of *Transformation Node.*

definition 5 (Transformation Node) *is a pair*

$$(\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is}), \mathbf{SizeAssigment}),$$

where $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$ *is a PTL which is the label of the node.* **SizeAssigment** *is a list of* **n** clause assignments, **n** *being the number of clauses in predicate* **Pred**. *Each such assignment refers to a different clause of* **Pred**, *and is a list of* **m** *items, where* **m** *is the cardinality of* **Os**. *There is an item for each argument number in* **Os**. *Each such item is a pair:*

$$(\mathbf{ArNum}, \mathbf{VSE}),$$

This pair describes the size of the term appearing at position number **ArNum** *of the clause head (denoted* $\mathbf{head}[\mathbf{ArNum}]$*) in relation with the sizes of other terms appearing at some clause positions.* **ArNum** *is an argument number,* $\mathbf{ArNum} \in \mathbf{Os}$, **VSE** *is a valid size expression and:*

1. $\mathbf{head}[\mathbf{ArNum}] = \mathbf{SR}(\mathbf{VSE})$ *is a valid size relation. Where* $\mathbf{SR}(\mathbf{VSE})$ *is a size relation obtained from* **VSE** *by replacing the* **TSDs** *which does not appear in other* **TSD** *by the term* $\mathbf{size}(\mathbf{Term})$, **Term** *being the term occurring at the position indicated by the* **TSD**. *E.g.* $\mathbf{SR}(1 + \mathbf{tsd}(\mathbf{ptl}(\mathbf{append}/3, [3], [2]), 3, 1, [\mathbf{h}(2)])) = 1 + \mathbf{size}(\mathbf{R})$ *(Note that* $\mathbf{size}(\mathbf{R}) \equiv \mathbf{body}_1[3]$*), and* $\mathbf{head}[3] = 1 + \mathbf{size}(\mathbf{R})$ *is a valid size relation for the recursive clause of the predicate* `append`*/3.*

2. *All head positions appearing in the size expressions of* **SizeAssigment** *are in* **Is**.

3. *If*

   $$\mathbf{tsd}(\mathbf{ptl}(\mathbf{PTL}1, \mathbf{ArNum}1, \mathbf{LitNum}1, \mathbf{SizeExp}1)) \text{ and}$$
   $$\mathbf{tsd}(\mathbf{ptl}(\mathbf{PTL}2, \mathbf{ArNum}2, \mathbf{LitNum}2, \mathbf{SizeExp}2))$$

   are two term Size descriptors appearing in any clause assignment, and $\mathbf{LitNum}1 = \mathbf{LitNum}2$, *then*

   $$\mathbf{PTL}1 = \mathbf{PTL}2 \text{ and } \mathbf{SizeExp}1 = \mathbf{SizeExp}2.\square$$

Condition 2 states that all the term sizes that are needed from a clause head are actually supplied by it.

Condition 3 states that a body literal can only be transformed in one way, and that the sizes supplied to it can be computed also in only one way.

Example 4.1 Consider Example 3.1, where a procedure transformation is proposed for the predicate `append/3`. The information needed for this transformation can be represented with the following transformation node:

```
(ptl(append/3,[3],[2]),
  [ [(3,h(2)) ],
    [(3,1+tsd(ptl(append/3,[3],[2]),3,1,[h(2)]))]]).
```

The procedure transformation process is trivial given this information. □

The intuition which can be gathered from the previous example is that it is possible to perform the size computation at run-time if some conditions hold on the transformation nodes. This will be the subject of the following sections.

5 Transforming Sets of Procedures: *Transformations*

In this section we deal with the problem of transforming sets of procedures with a callee-caller relationship, in order that they perform a size computation. In this case it is necessary to have at least a transformation node for some of them and these nodes have to meet some conditions that are explained below. To define the concept of *Transformation*, which informally can be considered as the information needed to transform a set of procedures, we need the following definitions:

definition 6 (*Con* relation) *We define a relation,* **Con** *between transformation nodes as follows:*

$(\mathbf{N}_1, \mathbf{N}_2) \in \mathbf{Con}$ *if and only if the label of* $\mathbf{N}_2$, $\mathbf{PTL}_2$ *appears in some term size descriptor of the size expressions of* $\mathbf{N}_1$, *i.e. there is a TSD in* $\mathbf{N}_1$ *of the form:*

$$\mathbf{tsd}(\mathbf{ptl}(\mathbf{PTL}_2, \mathbf{ArNum}, \mathbf{LitNum}, [\mathbf{Exp1}, .., \mathbf{ExpN}])$$ □

definition 7 (Connected nodes) *Given a transformation node* **EP** *and a set of transformation nodes,* **TNS**, *where* $\mathbf{EP} \in \mathbf{TNS}$, *we define the set of connected transformation nodes,* $\mathbf{CN}(\mathbf{EP}, \mathbf{TNS})$ *as:*

$$\mathbf{CN}(\mathbf{EP}, \mathbf{TNS}) = \{\mathbf{N} \in \mathbf{TNS} \mid (\mathbf{EP}, \mathbf{N}) \in \mathbf{Con}^{\mathrm{T}}\},$$

where $\mathbf{Con}^{\mathrm{T}}$ *is the transitive closure of* **Con**.□

definition 8 (Ordering between PTLs) *Given two PTLs,*

$$\mathbf{X} = \mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is_x}) \text{ and } \mathbf{Y} = \mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is_y}),$$

we say that $\mathbf{X} < \mathbf{Y}$ *if and only if* $\mathbf{Is_x} \subset \mathbf{Is_y}$. □

For example: $\mathbf{ptl}(\mathbf{append}/3, [3], [2]) < \mathbf{ptl}(\mathbf{append}/3, [3], [1, 2])$, but: $\mathbf{ptl}(\mathbf{append}/3, [3], [2]) \not< \mathbf{ptl}(\mathbf{append}/3, [3], [1])$

definition 9 (Transformation) *A pair* $(\mathbf{VTN}, \mathbf{EP})$, *where* **EP** *is a transformation node, and* **VTN** *is a set of transformation nodes, is a* Transformation *if and only if:*

1. $\mathbf{EP} \in \mathbf{VTN}$.
2. *Let* $\mathbf{NS} = \{\mathbf{EP}\} \cup \mathbf{CN}(\mathbf{EP}, \mathbf{VTN})$, *then:*

   For each term size descriptor:

   $$\mathbf{tsd}(\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is}), \mathbf{ArNum}, \mathbf{LitNum}, [\mathbf{Exp1}, .., \mathbf{ExpN}])$$

   appearing in the size expressions of the nodes in **NS** *there is a transformation node in* **NS** *labeled with* $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$.

EP *is called the* entry point *of the transformation.* □

Example 4.1 constitutes a transformation, where the entry point is the node itself.

Example 5.1 Consider the predicate qsort/2 defined as:

```
qsort([],[]).
qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        qsort(Ls,Ls2),qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).

partition(F,[],[],[]).
partition(F,[X|Y],[X|Y1],Y2) :- X =< F,partition(F,Y,Y1,Y2).
partition(F,[X|Y],Y1,[X|Y2]) :- X > F,partition(F,Y,Y1,Y2).
```

Let be $\mathbf{N}_1$ the transformation node:

```
(ptl(qsort/2,[2],[]),
       [ [(2,0)],
         [(2,tsd(ptl(append/3,[3],[2]),3,4,
                  [1+tsd(ptl(qsort/2,[2],[]),2,3,[])]))]]).
```

Let be $\mathbf{N}_2$ the transformation node from Example 4.1, then, the pair $(\{\mathbf{N}_1, \mathbf{N}_2\}, \mathbf{N}_1)$ is a transformation, with entry point the node $\mathbf{N}_1$ □

definition 10 (Size Computation Specification (SCS)) *We define a* Size Computation Specification (SCS) *as a pair* $(\mathbf{Pred}, \mathbf{Os})$, *where* **Pred** *is the name and arity of the predicate to be transformed, and* **Os** *is a list of argument numbers whose sizes are computed by the transformed predicate at run-time.* □

definition 11 (Transformation for a SCS) *A Transformation for a SCS* $(\mathbf{Pred}, \mathbf{Os})$, *is a transformation* $(\mathbf{T}, \mathbf{EP})$ *such that the label of* **EP** *is of the form* $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$. □

theorem 1 *If there is a Transformation* $(\mathbf{T}, \mathbf{EP})$, *for a size computation specification* $(\mathbf{Pred}, \mathbf{Os})$, *such that the label of* **EP** *is* $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$, *then it is possible to transform the clauses of* **Pred** *to obtain a transformed Predicate* $\mathbf{Pred}'$, *such that* $\mathbf{Pred}'$ *computes the sizes of the arguments indicated in* **Os**, *provided that the sizes of arguments indicated in* **Is** *are supplied, besides of course performing the same computations that* **Pred** *does.*□

Section 6 discusses how we can choose the best transformations.

A note on the generation and nature of transformation nodes: this generation is performed through a mode analysis to determine data flow patterns [5, 7, 17, 18, 3] and an argument size analysis [8]. It is important to note that this combined analysis can in some cases infer direct size relations between arguments of a predicate. For example, it is possible to infer, for the predicate `append/3`, that the length of its third argument is the sum of its two first arguments, i.e. $\mathbf{h}(3) = \mathbf{h}(1) + \mathbf{h}(2)$. This information can be used to generate transformation nodes which can form part of a transformation, but which need to traverse less data because a size computation can be performed directly in one operation, rather than by counting during the execution of the predicate. Thus, at one program point we may decide to perform an arithmetic operation, provided that the needed sizes are known, or make a literal perform size computation by transforming it to perform data traversal.

6 Irreducible/Optimal Transformations

We are interested in transformations for SCSs having the minimum number of transformation nodes and each of them having the minimum number of items in **Is**, where $\mathbf{ptl}(\mathbf{Pred}, \mathbf{Os}, \mathbf{Is})$ is the PTL of any node in the transformation. That is, to transform a predicate to make it compute the sizes of some of its arguments, we would like to know which are the arguments whose sizes are strictly necessary to perform this computation (in order to add only the absolutely necessary additional arguments and operations to the transformed predicates) and also what is the minimum

number of predicates which have to be transformed. The problem of finding optimal irreducible transformations lies in the fact that we need to use two parameters (number of transformation nodes and number of arguments needed) in the comparison and some transformations may be incomparable, in the sense that one is smaller than the other one on one criteria but the converse is true on the other criteria. We first introduce the concept of *irreducible transformation* and show that to determine whether it is possible to transform a predicate, we only need to find irreducible transformations. Then we present some ideas regarding the obtention of optimal irreducible transformations.

definition 12 (Irreducible Transformation) *A transformation* $(\mathbf{T}, \mathbf{EP})$*, is* Irreducible *if and only if:*

1. *There is only one transformation node in* $\mathbf{T}$ *labeled with the same procedure transformation label.*
2. $\mathbf{T} = \{\mathbf{EP}\} \cup \mathbf{CN}(\mathbf{EP}, \mathbf{T})$
3. *There are no two transformation nodes in* $\mathbf{T}$*, labeled with the* **PTLs** $\mathbf{X}$ *and* $\mathbf{Y}$ *respectively, such that* $\mathbf{X} < \mathbf{Y}$*.* □

The transformation shown in Example 5.1 is irreducible.

theorem 2 *If there is a transformation* $\mathbf{T}$ *for a SCS,* $\mathbf{X}$*, then there is an irreducible transformation* $\mathbf{T}'$ *for* $\mathbf{X}$*.*□

Theorem 2 means that we only need to find irreducible transformations to determine whether a procedure is transformable to compute sizes. Obviously irreducible transformations have a smaller number of nodes than the transformations they have been obtained from, which will result in transformed procedures with potentially less overhead at run-time.

If we are interested for example in having transformations with minimal labels, according to the ordering defined previously between PTLs, then we may define the following order relation between transformation nodes:

definition 13 (Order relation between Transformation Nodes) *Let be* **TS** *the set of all irreducible transformations for a given SCS* $\mathbf{X}$*, and* $\mathbf{TN} = \bigcup_{(\mathbf{t},\mathbf{e}) \in \mathbf{TS}} \mathbf{t}$*.*

Given two transformation nodes $\mathbf{N_x} \in \mathbf{TN}$ *and* $\mathbf{N_y} \in \mathbf{TN}$*, labeled with* $\mathbf{X}$ *and* $\mathbf{Y}$ *respectively, we say that* $\mathbf{N_x} < \mathbf{N_y}$ *if and only if* $\mathbf{X} < \mathbf{Y}$*.* □

definition 14 (Minimal Transformation) *We say that a transformation* $(\mathbf{T_x}, \mathbf{E_x})$ *for a SCS* $\mathbf{X}$ *is minimal if and only if for each node* $\mathbf{N_x} \in \mathbf{T_x}$ *there is no another transformation* $(\mathbf{T_y}, \mathbf{E_y})$ *for* $\mathbf{X}$*, which has a node* $\mathbf{N_y} \in \mathbf{T_y}$ *such that* $\mathbf{N_y} < \mathbf{N_x}$*.* □

theorem 3 *Let* **TS** *be the set of all irreducible transformations for a given SCS* $\mathbf{X}$*. If* $\mathbf{TS} \neq \emptyset$*, then there is a non-empty set of irreducible transformations* **MS**, $\mathbf{MS} \subseteq \mathbf{TS}$*, which contains only minimal transformation nodes.*□

Theorem 3 is interesting in that it provides us with a set of minimal irreducible transformations under the given order between transformation nodes. However, it may still be the case that there are other irreducible transformations which result in less overhead when performing data size computation at run-time, because a combination of having a smaller number of nodes and a smaller number of overall extra arguments. In general, we want to do predicate transformations which traverse the minimum amount of data. Thus we need to have a criterion to evaluate irreducible transformations in order to decide which of them will have the least overhead at run-time. To do this we may define other order relations. An interesting approach may be to obtain time cost functions for each irreducible transformation, by applying complexity analysis techniques, and to compare them.

7 Searching for Irreducible Transformations

Since the number of transformation nodes for a given SCS is finite, a possible algorithm to find transformations may be to simply generate all possible sets of transformation nodes and test which of them are irreducible transformations. However, some other more efficient approaches are possible.

One possible approach is to follow a top-down algorithm. This approach is based on the generation of *AND-OR* trees, where PTLs are the OR nodes and transformation nodes are the AND nodes. The search process is similar to SLD-Resolution. In this analogy, we can regard the resolvent in our *SLD-Like* algorithm, as the set of PTLs for which it is necessary to find transformation nodes labeled with them. Our current substitution, which we call *current transformation*, is the set of transformation nodes assigned to PTLs, and it will constitute the *answer transformation*. Thus, when the resolvent is empty the current transformation is the answer transformation, which will be irreducible. The entry point of an answer transformation is the transformation node assigned to the PTL that constitutes the root of the search tree. We represent the current transformation as a list of transformation nodes. Since there may be several PTLs for a given SCS, it is necessary to generate several search trees, with each PTL being the root of each tree. The search process starts with the resolvent being a PTL, which is the root of the tree, and an empty current transformation. A resolution step consists of removing a PTL from the resolvent and assigning to it a transformation node which is labeled with this PTL and it does not contain PTLs in its size expressions that are greater than some label of the nodes in the current transformation. This transformation node will be added to the current transformation. After this the resolvent is modified, by adding all the PTLs that appear in the selected transformation node such that a) they are not yet in the resolvent and b) no identical transformation node appears in the current transformation labeled with such PTLs.

Once we get all the answer irreducible transformations of all the possible *AND-OR trees*, we may decide which of them will have the least overhead in the size computation process.

The efficiency of the previous top-down algorithm can be improved if the alternatives for the *OR nodes* are generated with some knowledge regarding which PTLs will fail. If the base cases of recursive predicates are examined, it is possible to ensure that some PTLs will fail, and prune the search trees considerably. That is, apply a top-down driven bottom-up algorithm.

Another alternative is to apply directly a bottom-up algorithm. In this approach, first transformation nodes are found for the leaves in the call-graph, and this information is propagated to find transformation nodes for the ancestors, until we get to the root. Finding a transformation node will imply in this approach the computation of a *fixed-point*.

Example 7.1 Consider the predicate `qsort/2` as defined in Example 5.1, and suppose we want to transform it to compute the length of its second argument. We can apply a top-down algorithm. To do this we need to generate some transformation nodes. Consider for example $\mathbf{N}_1$, $\mathbf{N}_2$ and $\mathbf{N}_3$, where:

$\mathbf{N}_1$ is:

```
(ptl(qsort/2,[2],[]),
  [[(2,0)],
   [(2,tsd(ptl(append/3,[3],[]),3,4,[]))]]).
```

$\mathbf{N}_2$ is:

```
(ptl(qsort/2,[2],[]),
       [ [(2,0)],
         [(2,tsd(ptl(append/3,[3],[2]),3,4,
                  [1+tsd(ptl(qsort/2,[2],[]),2,3,[])]))]]).
```

and $\mathbf{N}_3$ is:

```
(ptl(append/3,[3],[2]),
 [  [ (3,h(2)) ],
    [ (3,1+tsd(ptl(append/3,[3],[2]),3,1,[h(2)])) ]]).
```

We can generate a tree for each possible PTL, but in this example we are going to generate one for **ptl**(**qsort**/2, [2], []). Thus, the first step is to initialize the resolvent with this label obtaining the initial state:

Resolvent: [**ptl**(**qsort**/2, [2], [])], and Current Transformation: []

Then we remove this PTL from the resolvent to find a transformation node labeled with it. We first choose $\mathbf{N}_1$, and add it to the *current transformation list*. After this, the resolvent is modified by adding **ptl**(**append**/3, [3], []) to it. The label **ptl**(**qsort**/2, [2], []) is not added to it because there is still a node in the current transformation labeled with it. At this point the current state is:

Resolvent: [**ptl**(**append**/3, [3], [])], and Current Transformation: [$\mathbf{N}_1$]

Then we remove the PTL from the resolvent and try to find a transformation node labeled with it. But because there is no such node, failure occurs and backtracking is performed, so that the new state is:

Resolvent: [**ptl**(**qsort**/2, [2], [])], and Current Transformation: []

We proceed and find another alternative for **ptl**(**qsort**/2, [2], []), which is $\mathbf{N}_2$, reaching the state:

Resolvent: [**ptl**(**append**/3, [3], [2])], and Current Transformation: [$\mathbf{N}_2$]

The next step is to find a node labeled with **ptl**(**append**/3, [3], [2]). This node is $\mathbf{N}_3$. At this point the resolvent is empty, and the current transformation is an irreducible transformation. The final state is:

Resolvent: [], and Current Transformation: [$\mathbf{N}_2, \mathbf{N}_3$]

This search may continue until find all possible transformations with entry point labeled with **ptl**(**qsort**/2, [2], []) are found. Moreover, all possible search trees can be generated from the possible labels referred to the SCS (**qsort**/2, [2]) as its root. □

8 Advantages of the Predicate Transformation Approach to Compute Sizes

As mentioned in the introduction, the standard approach to computing data sizes is to introduce new calls to predicates that explicitly compute them. For example, we can use the Prolog *length*/2 built-in to compute lengths of lists or use other similar built-ins. However, this approach involves an overhead which includes both having to traverse significant parts of the term (often the entire term) and the counting process done during this traversal. The transformations that we propose result in programs which although they still in general perform the counting, avoid the additional term traversal overhead since it is embedded in the traversals done by predicates which already existed in the program. Also, calculation itself is obviated when possible.

Furthermore, note that a transformed predicate may in fact traverse less or smaller data to compute sizes than when using a built-in. Consider, for example, the predicate p/2 defined as follows:

```
p([],[]).
p([X|Y],[X,X,X|Y1]):- p(Y,Y1).
```

If we have the goal `(p(X,Y),...)`, with the first argument of `p/1` ground, and the second unbound, and we need to know the length of the second argument after its execution, the standar approach would include a call to `length/2` as follows: `(p(X,Y),length(Y,L),...)`. In this case `length(Y,L)` has to traverse a list of length three times greater than the length of `X`. However, if we transform `p/2`:

```
p2o1i([],[],0).
p2o1i([X|Y],[X,X,X|Y1],S):- p2o1i(Y,Y1,Sb), S is Sb + 3.
```

and call: `p2o1i(X,Y,L)` to compute `L`, the number of sums performed is equal to the length of `X`, that is, three times lower than with the previous solution. Of course, the converse may also be true in some cases, but then the traversal is done in any case already by the program.

Consider another case – let us assume that we have:

```
q(X), append(Y,X,Z), append(W,X,K)
```

where `X`, `Y` and `W` are ground lists, and `Z` and `K` are unbound variables that will be bound to lists when the goal succeeds. Let us also assume that we are interested in knowing the lengths of `Z` and `K` after the execution of the goal. Using the standard approach we may have:

```
q(X), append(Y,X,Z), append(W,X,K), length(Z,LZ), length(K,LK)
```

while using the predicate transformation approach we would have:

```
q1o(X,SX), append3o2i(Y,X,Z,SX,SZ), append3o2i(W,X,K,SX,SK)
```

where `q1o(X,SX)` computes the length of `X` (`SX`), which is used by `append3o2i/5` to compute the lengths of `Z` and `K` (`SZ` and `SK`). In this case the sum of the lengths of the data traversed, which is equivalent to the operations needed to compute the lengths is: $\mathbf{length(X)} + \mathbf{length(Y)} + \mathbf{length(W)}$ In the first case we have: $\mathbf{length(Z)} + \mathbf{length(K)}$, but since: $\mathbf{length(Z)} = \mathbf{length(X)} + \mathbf{length(Y)}$, and $\mathbf{length(K)} = \mathbf{length(X)} + \mathbf{length(W)}$ we have: $2 * \mathbf{length(X)} + \mathbf{length(Y)} + \mathbf{length(W)}$

One might think that a better solution to the first approach would be:

```
q(X), append(Y,X,Z), append(W,X,K), length(X,SX),
length(Y,SY), length(W,SW), SZ is SX + SY, SK is SX + SW
```

but in this case it is necessary to analyze the program to infer that the length of the third argument of `append/3` is the sum of its two first arguments. This may be easy in some cases, for example for `append/3`, but may be more difficult or impossible in some other cases. This is the case when the length of a list depends not only on the length of other lists but also on its contents. In any case, note that our approach would still take advantage of such optimizations if they can be detected.

9 An Application: Granularity Control

Dynamic term size computation has a important application in Granularity Control of Logic Programs [8]. Logic programming languages offer a great deal of scope for parallelism. It may in fact be possible to extract "maximal" parallelism for a program [14, 12, 6]. This is interesting in the abstract. However, just because something *can* be done in parallel does not necessarily mean, in practice, that it *should* be done in parallel. This is because the parallel execution of a task incurs various overheads, e.g. overheads associated with process creation and scheduling, the possible migration of tasks to remote processors and the associated communication overheads, etc. In general, a goal should not be a candidate for parallel execution if its granularity, i.e. the "work available" underneath it, is less than the work necessary to create a separate task for that goal. A number of researchers have investigated the automatic analysis of the (time) complexity of programs (see, for example, [2, 13, 15, 19, 20, 21, 22]). The Granularity Analysis we consider

here [8] differs from these in some aspects that we will not discuss for the sake of brevity, but which are centered around the fact that no execution of the program is required to perform the analysis. It provides granularity estimates that are an upper bound on the amount of work that may be done at runtime. Since the work done by a call to a recursive predicate typically depends on the size of its input, this technique consists in doing as much of the analysis at compile time as possible, but postponing the actual computation of granularity until runtime, when input data sizes are known. To do this computation and to decide whether to execute in parallel or in sequential, programs are transformed. This includes adding conditionals to clauses and some extra literals which compute input data sizes and perform cost estimations. This process can be done automatically for a class of programs.

Example 9.1 Consider a parallel version of the definition of the **qsort**/2 predicate given in Example 5.1:

```
qsort([],[]).
qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        qsort(Ls,Ls2)&qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

in which `qsort(Ls,Ls2)` and `qsort(Lg,Lg2)` are executed in parallel, as described by the & symbol [11, 9, 4]. The cost analysis performed at compile-time would provide a function that gives an upper-bound on the cost of predicate `qsort`/2 in terms of the size of its first argument, assuming that this argument is ground at procedure invocation. Then a predicate transformation can be done automatically for predicate `qsort`/2 in order to perform granularity control. As a result of this transformation the following code is obtained:

```
% Version of qsort/2 that performs granularity control.
g_qsort([],[],_).
g_qsort([First|L1],L2,Size1) :-
      % compute upper-bound of execution time.
      qsorttime(Size1,Time),
      Time < 10 ->
         (partition(First,L1,Ls,Lg),
          s_qsort(Ls,Ls2),s_qsort(Lg,Lg2));
         (trpartition(First,L1,Ls,Lg,SizeLs,SizeLg),
          g_qsort(Ls,Ls2,SizeLs)&g_qsort(Lg,Lg2,SizeLg)),
      append(Ls2,[First|Lg2],L2).
% Sequential version for qsort/2.
s_qsort([],[]).
s_qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        s_qsort(Ls,Ls2),s_qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

where the literal `qsorttime(Size1,Time)` computes an upper-bound of the cost of executing the clause body sequentially, evaluating the function inferred through analysis at compile-time. We have omitted it for the sake of conciseness. The constant `10` represents some experimentally determined threshold which is directly related to the cost of creating a parallel task. The literal `trpartition(First,L1,Ls,Lg,SLs,SLg)` is the transformed version of `partition(First,L1,Ls,Lg)`, that computes the sizes of its third and fourth arguments (`SizeLs` and `SizeLg` represent the sizes of `Ls` and `Lg` respectively). The definition of `trpartition`/5, which can be obtained automatically from the analysis herein presented, would be:

```
trpartition(F,[],[],[],0,0).
trpartition(F,[X|Y],[X|Y1],Y2,SL,SG) :-
```

```
        X =< F,
        trpartition(F,Y,Y1,Y2,SL1,SG), SL is SL1 + 1.
trpartition(F,[X|Y],Y1,[X|Y2],SL,SG) :-
        X > F,
        trpartition(F,Y,Y1,Y2,SL,SG1), SG is SG1 + 1.
```

□

We have presented Granularity Control as our application of dynamic term size computation but, in general, it can be applied in any case which needs exact values of sizes to make decisions at run-time, such as the reform compilation method mentioned in the introduction [16, 1] which dynamically unravels recursions provided the iteration lengths are known.

10 Conclusions and Future Work

We have described how predicates can be transformed to compute term sizes at run-time and pointed out the advantages of such transformation. We have also shown a top-down algorithm to find irreducible transformations, which we have implemented in its main part. We are planning on finishing this implementation and evaluating its performance in the granularity application that we have described. We also plan to search for new algorithms, and compare them in order to perform the predicate transformation in the most efficient way possible. This work is oriented to the development of a complete granularity control system, which can be considered the source of inspiration behind the dynamic term size computation technique presented. In this sense we are working on the integration of this system into a series of other program analysis an transformation tools, that we have implemented, in order to develop improved automatic parallelizing compilers for logic programs.

References

[1] Jonas Barklund and Håkan Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 817–824, ICOT, Japan, 1992. Association for Computing Machinery.

[2] B. Bjerner and S. Holmstrom. A compositional approach to time analysis of first order lazy functional programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.

[3] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[4] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[5] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[6] S. K. Debray. A Simple Code Improvement Scheme for Prolog. In *Sixth International Conference on Logic Programming*, pages 17–32. MIT Press, June 1989.

[7] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[8] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[9] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[10] P. López García and M. Hermenegildo. Towards Dynamic Term Size Computation via Program Transformation. Technical Report CLIP7/93.0, Computer Science Dept, Universidad Politecnica de Madrid, July 1993.

[11] M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[12] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 369–389. MIT Press, 1989.

[13] T. Hickey and J. Cohen. Automating Program Analysis. *Journal of ACM*, 35(1), Jan 1988.

[14] L. Kale. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[15] D. Le Métayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[16] Håkan Millroth. Reforming compilation of logic programs. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 485–502, San Diego, USA, 1991. The MIT Press.

[17] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[18] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[19] M. Rosendhal. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. MIT Press, 1989.

[20] P. Wadler. Strictness analysis aids time analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1988.

[21] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9), Sep 1975.

[22] S. Duvvuru L. Hansen A.V.S. Sastry X. Zhong, E. Tick and R. Sundararajan. A new method for compile time granularity analysis. In *ILPS'91 Workshop on Compilation of Symbolic Languages for Parallel Computers.*